High Performance Beowulf Cluster Environment

Hamilton Cluster

User Manual

Version 1.01

15th December 2005

Table of Contents

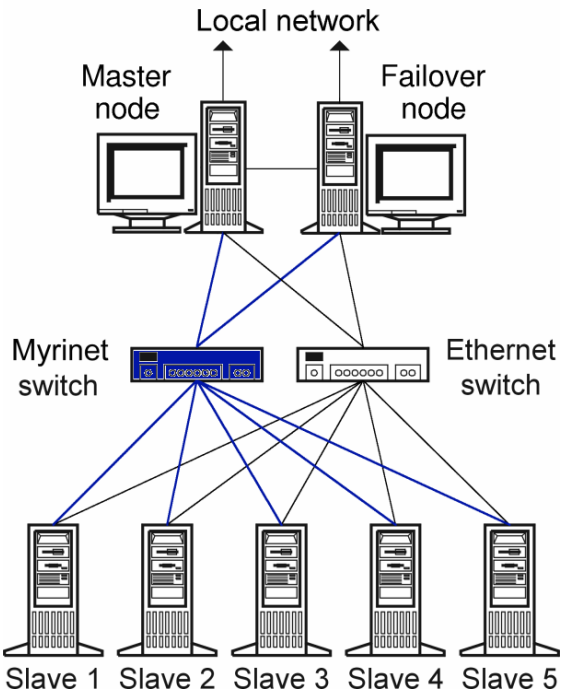
1 Introduction....4
2 The physical hardware layout of Hamilton....4
3 Getting an account on Hamilton....5
4 Accessing the cluster....5
5 The Hamilton cluster....6
6 The Modules Environment....7
7 Running your application on the cluster....8
8 Selecting the correct module environment....8
9 Compiling your code....10
10 MPI compile and link commands....11
11 Executing the program....11
11.1 Using MPICH....12
11.2 Using MPICH-GM....12
12 Running your program from the Grid Engine queuing system....13
12.1 Grid Engine....14
12.2 Sample batch submission script....14
12.3 Sample script for MPICH....15
12.4 Grid Engine commands....16
13 Software....18
13.1 BLAS, LAPACK, FFT and SCALAPACK libraries....18
13.2 Numerical Algorithms Group (NAG) Fortran Library....19
13.3 Matlab....19
13.4 Molpro....20
14 Creating your own modules (Advanced users)....20
15 An example MPI program....21
15.1 Enter and Exit MPI....23
15.2 Who Am I? Who Are They?....23
15.3 Sending messages....23
15.4 Receiving messages....23

1 Introduction

This guide introduces the ClusterVision Beowulf Cluster Environment running on the Hamilton Cluster at the University of Durham ('hamilton'). It explains how to use the MPI and batch environments, how to submit jobs to the queuing system and how to check your job progress. If you are short of time and would like a quick guide please see quick.sxw.

2 The physical hardware layout of Hamilton

Hamilton is an example of a Beowulf Cluster and consists of a login, compile and job submission node, called the master node, and 135 compute nodes, normally referred to as slave nodes. A second (fail over) master is available in case the main master node fails. Each of the slave nodes consists of two 2 GHz AMD Opteron processors and 4 Gbytes of memory. An additional second fast network called a 'Myrinet' is available for fast communication between 32 of the 135 slave nodes (see figure).



The master node is used to compile software, to submit a parallel or batch program to a job queuing system and to analyse produced data. Therefore, it should rarely be necessary for a user to login to one of the slave nodes.

The master node and slave nodes communicate with each other through a Gigabit Ethernet network. This can send information at a maximum rate of 1000 Mbits per second.

The additional *Myrinet* network is added to the cluster for faster communication between 32 of the slave nodes. This faster network is mainly used for MPI programs that can use multiple machines in parallel in order to solve a particular computational problem and need to exchange large amounts of information. The *Myrinet* network can send information at a maximum rate of 4000 Mbits per second.

The AMD Opteron CPUs are 64 bit and are referred to as x86_64 architecture. This architecture allows addressing of large memory sizes. However, one of the additional advantages of Opterons is that they can also run executables built for Intel (x86) 32-bit processors (known as x86_32) such as ITS linux computers like vega.

3 Getting an account on Hamilton

If you already have an account on `hal` you will be able to use `hamilton` under the ITS project allocation. If you do not already have an account please apply by submitting the application form available from
http://www.dur.ac.uk/resources/its/helpdesk/registration/forms/HighPerfRegistration.pdf
to the ITS helpdesk. You account will be ready for use the next day with the same password as your standard ITS account.

4 Accessing the cluster

The master node is called hamilton. From outside the cluster, it is known by this name only.

If you wish to access the master node use `ssh` (secure shell). The command structure for ssh is:

```
ssh user@hostname
```

So, if you are user 'dxy0abc' then the command

```
ssh dxy0abc@hamilton
```

will log you into the cluster. From outside Durham it would be `hamilton.dur.ac.uk`. If you wish to use X11 graphics the -Y option requests X11 forwarding and for programs such as qmon or `nedit`:

```
ssh -Y dxy0abc@hamilton
```

From a computer running Windows we would recommend using PuTTY (which can be obtained from

http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html).

5 The Hamilton cluster

Once you login you will be in your ITS home space (`/users/username`). Files here are stored on the main ITS file servers (`hudson` and `stevens`) and are backed up overnight, but there is limited space available. This is where you should store your source code and other important files. Each user is given space on the master node in a directory called `/data/hamilton`. Your space will be in the directory with your username,
e.g. `/data/hamilton/username`. Please use this directory to run jobs from as it is close to the slave nodes and will place less load on the main file servers. Files will not be deleted from here but are not backed up. The filestore is a RAID device however.

You can also access the ITS scratch space `/scratch/share`. This is accesible from all ITS Unix or Linux computers and can be used to transfer files or data across. Data here is not backed up and is deleted regularly.

Each of the slave nodes on the `hamilton` cluster has local scratch space available as /scratch/local. You can use this for temporary storage whilst your job runs on the slave node.

Locally installed software is in `/usr/local/packagename` (e.g. `/usr/local/dlpoly`) which is mounted from the file servers or on the master

itself under `/usr/local/Cluster-Apps/packagename` (e.g. `/usr/local/Cluster-Apps/castep`).

6 The Modules Environment

On a complex computer system with a choice of software packages and software versions it can be quite hard to set up the correct computing environment. For instance, managing different MPI software packages on the same system or even different versions of the same MPI software package is almost impossible on a standard Suse or Red Hat system as a lot of software packages use the same names for the executables and libraries. As a user you end up with the problem that you can never be quite sure which libraries are used for the compilation of a program as multiple libraries with the same name might be available. Also, very often you would like to test new versions of a software package before permanently installing it. Within Red Hat or Suse this would be a quite complex task to achieve. The `module` environment makes this process easy.

The command to use is `module`:

`module`	
`(no arguments)`	print usage instructions
`avail`	list available software modules
`load modulename`	add a module to your environment
`add modulename`	add a module to your environment
`unload modulename`	remove a module
`clear`	remove all modules

This is an example output from `module avail`:

```
------------------/usr/local/Cluster-Config/Modulesfiles/------------------
castep/3.0 ..............molpro/2002.6           mpich_gm/1.2.5..12-pgi522
default-ethernet          mpich/1.2.6-32b-pgi522    null
default-myrinet           mpich/1.2.6-gcc333        pathscale/1.2
dot                       mpich/1.2.6-pathcc12      pgi/5.2-2
gm/2.0.13                 mpich/1.2.6-pgi522        sge/6.0u1
module-info               mpich_gm/1.2.5..12-gnu333 use.own
modules                   mpich_gm/1.2.5..12-pathcc version
```

This indicates that there are several different modules available on the system including application software (e.g. castep), compilers (e.g. pathscape, pgi) and MPI parallel environments (e.g. mpich and mpich_gm). To get more information on available modules try typing `module help` or `module whatis`.

There is a default module available called `default-ethernet` which loads several other modules, namely the Portland compilers, Grid Engine (sge: the queue manager) and mpich for the ethernet. To see what modules you have currently loaded type `module list`

```
Currently Loaded Modulefiles:
  1) /null                 4) /modules              7) /default-ethernet
  2) /dot                  5) /mpich/1.2.6-pgi522
  3) /sge/6.0u1            6) /pgi/5.2-2
```

If you want to remove any of these you can run `module rm name`.To clear all modules type `module purge` and you can then load other modules. For example, if you wanted to use the Myrinet interface you would load the `default-myrinet` module.

7 Running your application on the cluster

In order to use and run your application on the cluster, four steps are necessary:

- selecting the correct module environment;
- compiling your application;
- creating an batch script for the queuing system; and
- submitting your application to the queuing system.

8 Selecting the correct module environment

As mentioned above the module system is used to select from a number of possible running environments. If you are running single processor serial jobs or parallel jobs on the Gigabit ethernet then the default environment 'default-ethernet' should be sufficient and you can skip this part. If you want run parallel

code on the Myrinet interface or change compilers, you will have to change from the default module.

There are two different MPI libraries on the cluster: MPICH for Ethernet based communication and MPICH-GM for Myrinet.

For each network system there is a choice of compilers: the open source GNU gcc, or the higher performance compilers from Portland or PathScale.

In order to select between the different compilers and MPI libraries, you should use the `module` environment. During a single login session, use the command-line tool `module`. This makes it easy to switch between versions of software, and will set variables such as the `$PATH` to the executables. However, such changes will only be in effect for this login session, and subsequent sessions will not use these settings.

To select a non-default combination of compiler and MPI library on a permanent basis you must edit your `.login` file. You first need to get rid of the deafult setup and then add in your new set-up. For example to use the default setup for the Myrinet you would add:

```
module purge
module add default-myrinet
```

to the end of the file. To use the PathScale compilers together with the gigabit ethernet you would add:

```
module purge
module add pathscale
module add mpich/1.2.6-pathcc12
```

The naming scheme for this combination can be explained as follows. The first part of the name is the type of MPI library, `mpich` for MPI over gigabit ethernet (it would be `mpich-gm` for MPI over Myrinet). This is followed by the version number of the MPI library (1.2.6). The last part of the name is the compiler used when the mpicc command is run (pathcc – the PathScale compilers).

9 Compiling your code

There are three compilers groups available on the master node; GNU, Portland Group (http://www.pgroup.com/) and PathScale (http://www.pathscale.com/) compilers. The Portland compilers have 5 floating licences and are recommended for general use. We have one PathScale licence which is on trial. The following table summarises the compiler commands available on hamilton:

Language	GNU compiler	Portland compiler	PathScale compiler
C	gcc	pgcc	pathcc
C++	c++	pgCC	pathCC
Fortran77	f77	pgf77	pathf90
Fortran90		pgf90	pathf90

The most common code optimisation flag for the GNU compiler is `-O3`. For the Portland compiler the `-fast` option expands out to basic compiler optimsation flags (type `pgf90 -fast -help` for more details). For the PathScale compilers use `-O`. For maximum application speed it is recommended to use the Portland or PathScale compilers. Please refer to the respective man-pages for more information about optimisation for GNU, Portland and PathScale. HTML and PDF documentation may be found in the `/usr/local/Cluster-Docs` directory.

By default the compiler will try to compile and link in one step. So, for example if you have a Fortran90 file called `monte.f90` you would type

```
pgf90 -o monte monte.f90
```

to compile and link it in one step to produce the executable `monte`. If you wish to compile and link in different steps you would use the `-c` option to the compiler to prevent the link stage:

```
pgf90 -c monte.f90
```

which would produce an object file `monte.o`. You could then link it in a separate step:

```
pgf90 -o monte monte.o
```

10 MPI compile and link commands

The commands referred to in the table above are specific for serial (single processor) applications. For parallel applications it is preferable to use MPI based compilers. The correct compilers are automatically available after choosing the parallel environment. The following MPI compiler commands are available:

Code	Compiler:
C	mpicc
C++	mpiCC
Fortran77	mpif77
Fortran90	mpif90

These MPI compilers are 'wrappers' around the GNU, Portland and PathScale compilers and ensure that the correct MPI include and library files are linked into the application (Dynamic MPI libraries are not available on the cluster). Since they are wrappers, the same optimisation flags can be used as with the standard GNU or Portland compilers.

Typically, applications use a Makefile that has to be adapted for compilation. Please refer to the application's documentation in order to adapt the Makefile for a Beowulf cluster. Frequently, it is sufficient to choose a Makefile specifically for a Linux MPI environment and to adapt the `CC` and `F77` parameters in the Makefile. These parameters should point to `mpicc` and `mpif77` (or `mpif90` in the case of Fortran 90 code) respectively.

11 Executing the program

There are two methods for executing a parallel or serial batch program: using the queuing system or directly from the command line. In general you should use the queuing system, as this is a production environment with multiple users and example scripts which you can copy are discussed below. However, if a quick test of the application is necessary you can use the command line and run outside the queuing system on the master node, but only with 1 or 2 processes.

To run a serial program such as the executable `monte` above you only need to type the name of the executable to run it:

```
monte
```

All installed parallel MPI environments need to know on what slave nodes to run the program. The methods for telling the program which nodes to use differ however.

11.1 Using MPICH

The command line for MPICH would look like this:

```
mpirun -machinefile configuration_file -np 4 program_name program_options
```

Where the configuration file is chosen by the Grid Engine software and looks something like this:

```
node02:2
node03:2
```

The `:2` extension tells MPICH that it is intending to run two processes on each node. Please refer to the specific man-pages and the command `mpirun -h` for more information.

11.2 Using MPICH-GM

A typical command line for MPICH-GM (Myrinet based communication) in the directory where the program can be found is the following:

```
mpirun.ch_gm --gm-kill 1 --gm-f configuration_file -np 4 program_name program_options
```

The `configuration_file` consists typically of the following lines:

```
node02
node02
node03
node03
```

The `configuration_file` chosen by Grid Engine shows that the application using this configuration file will be started with two processes on each node, as in the case of dual CPU slave nodes.

The `-np` switch on the `mpirun.ch_gm` command line indicates the number of processes to run, in this case 4.

The list of options is:

```
mpirun.ch_gm [--gm-v] [-np <n>] [--gm-f <file>] [--gm-h] prog [options]
```

Option	Explanation
`--gm-v`	verbose - includes comments
`-np <n>`	specifies the number of processes to run
`--gm-np <n>`	same as `-np` (use one or the other)
`--gm-f <file>`	specifies a configuration file
`--gm-use-shmem`	enable the shared memory support
`--gm-shmem-file <file>`	specifies a shared memory file name
`--gm-shf`	explicitly removes the shared memory file
`--gm-h`	generates this message
`--gm-r`	start machines in reverse order
`--gm-w <n>`	wait `n` secs between starting each machine
`--gm-kill <n>`	`n` secs after first process exits, kill all other processes
`--gm-dryrun`	Don't actually execute the commands just print them
`--gm-recv <mode>`	specifies the recv mode, '`polling`', '`blocking`' or '`hybrid`'
`--gm-recv-verb`	specifies verbose for recv mode selection
`-tv`	specifies totalview debugger

Options `--gm-use-shmem` is highly recommended to use as it improves performance. Another recommended option is `--gm-kill`. It is possible for a program to get 'stuck' on a machine and MPI is unable to pick up the error. The problem is that the program will keep the port locked and no other program will be able to use that Myrinet port. The only way to fix this is to manually kill the MPI program on the slave node. If the option `--gm-kill 1` is used, MPI make a better effort to properly kill programs after a failure.

12 Running your program from the Grid Engine queuing system

A workload management system – also known as a batch or queuing system – allows users to make more efficient use of their time by allowing you to specify tasks to run on the cluster. It allows administrators to make more efficient use of cluster resources, spreading load across processors and scheduling tasks according to resource needs and priority.

The Grid Engine queuing system allows parallel and batch programs to be executed on the cluster. The user asks the queuing system for resources and the queuing system will reserve machines for execution of the program.

The user submits jobs and instructions to the queuing system through a shell script. The script will be executed by the queuing system on one machine only; the commands in the script will have to make sure that it starts the actual application on the machines the queuing system has assigned for the job. The system takes care of holding jobs until computing resources are available, scheduling and executing jobs and returning the results to you.

12.1 Grid Engine

Grid Engine is a software package for distributed resource management on compute clusters and grids. Utilities are available for job submission, queuing, monitoring and checkpointing. The Grid Engine software is made available by Sun Microsystems and the project homepage is at `http://gridengine.sunsource.net`, where extensive documentation can be found.

12.2 Sample batch submission script

This is an example script, used to submit non-parallel scripts to Grid Engine. This script can be found in `/usr/local/Cluster-Examples/sge/serial.csh`.

```
#!/bin/csh
#
# simple test script for submission of a serial job on hamilton
# please change the mail name below

# use current working directory
#$ -cwd
# combine error and output files
#$ -j y
# send email at start (s) and end (e)
#$ -m se
# email address
##$ -M my_mail_name@durham.ac.uk
# execute commands
myprog1
```

In order to submit a script to the queuing system, the user issues the command `qsub scriptname`. Usage of `qsub` and other queueing system commands will be discussed in the next section.

Note that `qsub` accepts only shell scripts, not executable files. All options accepted by the command-line `qsub` can be embedded in the script using lines starting with `#$` (see below).

The name of the job is set by default to the name of the script and the output will go into a file with this name with the suffix `.o` appended with the jobid, e.g. out put from the script with job number 304 will go into file serial.csh.o304.

The `-m` flag is used to request the user be emailed at start (s) and end (e) of the job. The job output is directed into the directory you are currently working from by `-cwd`. If you prefer to have standard error and standard output combined, leave the flag `-j y`. You can adda -P flag to specify which project the job should be charged from .e.g

#$ -P ITSall

When a job is submitted, the queuing system takes into account the requested resources and allocates the job to a queue which can satisfy these resources. If more than one queue is available, load criteria determine which node is used, so there is load balancing. There are several requestable resources, including system memory and max CPU time for a job for details type qstat -F.

12.3 Sample script for MPICH

This example script which can be found in `/usr/local/Cluster-Examples/sge/mpich-ethernet.csh` submits an MPICH job to run on the ethernet system.

```
#!/bin/csh
#
# simple test script for submission of a parallel mpi job
# using the gigabit ethernet on hamilton
# please edit the mailname below

# use current working directory
#$ -cwd
# combine error and output files
#$ -j y
# send email at start (s) and end (e)
```

```
#$ -m se
# email address
#$ -M my_mail_name@durham.ac.uk

# request between 2 and 4 slots
#$ -pe mpich 2-4

setenv MPICH_PROCESS_GROUP no

# set up the mpich version to use
# load the module

module purge
module load default-ethernet

# Gridengine allocates the max number of free slots and sets the
# variable $NSLOTS.
echo "Got $NSLOTS slots."

# execute command
mpirun -np $NSLOTS -machinefile $TMPDIR/machines  mympiexecutable
```

It requests from 2 to 4 parallel slots and runs the job. A similar file `mpich-myrinet.csh` can be used to submit jobs to run on the Myrinet network.

12.4 Grid Engine commands

The Grid Engine queuing system is used to submit and control batch jobs. The queuing system takes care of allocating resources, accounting and load balancing jobs across the cluster. Job output and error messages are returned to the user. The most convenient method to update or see what happens with the queuing system is using the X11-based command `qmon`.

The command `qstat` is used to show the status of queues and submitted jobs.

`qstat`	show job/queue status
`no arguments`	show currently running/pending jobs
`-f`	show full listing of all queues, their status and jobs
`-j [job_list]`	shows detailed information on pending/running job
`-u user`	shows current jobs by user

The command `qhost` is used to show information about nodes.

`qhost`	show job/host status
`no arguments`	show a table of all execution hosts and information about their configuration
`-l attr=val`	show only certain hosts

`-j [job_list]`	shows detailed information on pending/running job
`-q`	shows detailed information on queues at each host

The command `qsub` is used to submit jobs.

`qsub scriptname`	submits a script for execution
`-a time`	run the job at a certain time
`-l`	request a certain resource
`-q queues`	jobs is run in one of these queues

There are many other options to `qsub`, consult the man page for more information.

The command `qdel` is used to delete jobs.

`qdel jobid(s)`	deletes one (or more) jobs
`-u username`	deletes all jobs for that user
`-f`	force deletion of running jobs

So, to submit your job you would type

```
qsub serial.csh
```

and to check its status you can type

```
qstat
```

If there are lots of jobs in the system you might use the -u option together with your username to show only your jobs:

```
qstat -u dxy0abc
```

To get more detail about an individual job you can specify a job number:

```
qstat -j 304
```

You can use the qmon graphical user interface (GUI) to get information about your jobs and account status:

```
qmon &
```

To see the state of your account or project click on 'Policy Configuration' (fourth from left, bottom row) and then select “Share Tree Policy” at the bottom. You should then see a list of departments which you can click on to see groups and users. At the bottom is a list of the attributes of the highlighted group or user. The Shares entry is the number of shares allocated. The level percentage is the percentage that the allocated number of shares represents out of that level. So for example if you click on Chemistry you can see they have 15400 shares which is 23.5% of the resource at the departmental level (Level Percentage) and which represents 23.5% of the total resource (Total Percentage). Now you can also see Chemistry has three groups namely `chemmodel`, `coldmols` and `liquidcrystal`. Clicking on them indicates the relative distribution of the 15400 shares within Chemistry. If you click on say `chemmodel` you can see it has 4000 shares which is 26% of Chemistry's allocation and 6.1% of the total resource. The Actual Resource Share indicates the percentage the level has used so far.

The quick queue for short jobs

There is a quick queue called all.q set up for jobs requiring less than 12 hours CPU time. To run your job in this queue you should add

```
#$ -q all.q
#$ -l quick
#$ -l h_cpu=12:0:0
```

to your job submission script.

13 Software

13.1 BLAS, LAPACK, FFT and SCALAPACK libraries

The AMD Core Maths Libraries (ACML) incorporating BLAS (Basic Linear Algebra Subprograms; http://www.netlib.org/blas), LAPACK (Linear Algebra Package; http://www.netlib.org/lapack), FFT and SCALAPACK (http://www.netlib.org/scalapack) routines are installed in /usr/local/atlas/acml/2.5.0 . These routines have been tuned for the AMD Opteron processor. Documentation for the libraries is available in /usr/local/atlas/acml/2.5.0/Doc, e.g. /usr/local/atlas/acml/2.5.0/Doc/html/index.html or /usr/local/atlas/acml/2.5.0/Doc/acml.pdf and also at

http://www.pgroup.com/support/libacml.htm

To compile a program foo.f with the Portland compiler and link it with the acml library the following command may be used:

```
pgf90 foo.f -o foo -Mcache_align -tp k8-64 \
-I/usr/local/atlas/acml/2.5.0/pgi64/include \
/usr/local/atlas/acml/2.5.0/pgi64/lib/libacml.a
```

13.2 Numerical Algorithms Group (NAG) Fortran Library

This is installed in /usr/local/nag/current. Documentation is available in /usr/local/nag/current/doc. An overview is provided by pointing mozilla at the User Note:

```
mozilla /usr/local/nag/current/doc/un.html
```

The nagexample script demonstrates use of the library e.g.

```
nagexample x02ajf
```

and you can compile a fortran program thus:

```
pgf77 x02ajfe.f /usr/local/nag/current/lib/libnag.a \
/usr/local/nag/current/acml/libacml.a
```

13.3 Matlab

Matlab commands in a matlab file Untitled.m can be run using the following command in a serial batch file (such as /usr/local/Cluster-Examples/sge/serial.csh):

```
module add matlab
matlab -nodisplay < Untitled.m
```

13.4 Molpro

Single processor molpro jobs (e.g in a file called h2o_eom.test) can be run by adding the following commands to a serial batch file (such as /usr/local/Cluster-Examples/sge/serial.csh):

```
module add molpro
molpros_2002_6 -v < h2o_eom.test
```

14 Creating your own modules (Advanced users)

It is possible to create new modules and add them to the module environment. After installing an application in (for instance) `~/my_application` and creating a new module directory in `~/my_modules` you can modify the `MODULEPATH` to include the new search path. If you use the tcsh shell (the default) the command would be

```
setenv MODULEPATH $MODULEPATH:~/my_modules
```

If you use the bash shell this command would be:

```
export MODULEPATH=$MODULEPATH:~/my_modules
```

To make the change permanent, please add this command to your `.cshrc` or `.bashrc` file. The contents of a module look like this:

```
##%Module1.0#####################################
##
## mpich-gnu modulefile
##
proc ModulesHelp { } {

        puts stderr "\tAdds myrinet mute to your environment"
}

module-whatis   "Adds myrinet mute to your environment"

set             root                    usr/local/Cluster-Apps/mute-1.9.5
setenv          GM_HOME                 $root
setenv          gm_home                 $root
append-path     PATH                    $root/bin/
append-path     MANPATH                 $root/man
append-path     LD_LIBRARY_PATH         $root/lib/
```

Typically you only have to fill in the root path (`/usr/local/Cluster-Apps/mute-1.9.5`) and the description to make a new and fully functional module. For more information, please load the 'modules' module (module load modules), and read the module and modulefile man pages. The best policy is to make an additional directory underneath the modules directory for each application and to place your new module in there. Then you can change the name of the module file such that it reflects the version number of the application. For instance this is the name of the location of the `mute` module: `/usr/local/Cluster-Config/Modulefiles/mute/1.9.5` Now by issuing the command 'module load mute' you will automatically load the new module. The advantage is that if you have different version of the same application, this command will always loads the most recent version of that application.

15 An example MPI program

The sample code below contains the complete communications skeleton for a dynamically load balanced master/slave application. Following the code is a description of the few functions necessary to write typical parallel applications.

```
#include <mpi.h>
#define WORKTAG     1
#define DIETAG     2
main(argc, argv)
int argc;
char *argv[];
{
       int         myrank;
       MPI_Init(&argc, &argv);   /* initialize MPI */
       MPI_Comm_rank(
       MPI_COMM_WORLD,   /* always use this */
       &myrank);      /* process rank, 0 thru N-1 */
       if (myrank == 0) {
              master();
       } else {
              slave();
       }
       MPI_Finalize();       /* cleanup MPI */
}

master()
{
       int     ntasks, rank, work;
       double       result;
       MPI_Status     status;
       MPI_Comm_size(
       MPI_COMM_WORLD,   /* always use this */
       &ntasks);          /* #processes in application */
/*
* Seed the slaves.
*/
```

```
       for (rank = 1; rank < ntasks; ++rank) {
              work = /* get_next_work_request */;
              MPI_Send(&work,         /* message buffer */
              1,              /* one data item */
              MPI_INT,        /* data item is an integer */
              rank,           /* destination process rank */
              WORKTAG,        /* user chosen message tag */
              MPI_COMM_WORLD);/* always use this */
       }

/*
* Receive a result from any slave and dispatch a new work
* request work requests have been exhausted.
*/
       work = /* get_next_work_request */;
       while (/* valid new work request */) {
              MPI_Recv(&result,       /* message buffer */
              1,              /* one data item */
              MPI_DOUBLE,     /* of type double real */
              MPI_ANY_SOURCE, /* receive from any sender */
              MPI_ANY_TAG,    /* any type of message */
              MPI_COMM_WORLD, /* always use this */
              &status);       /* received message info */
              MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
              WORKTAG, MPI_COMM_WORLD);
              work = /* get_next_work_request */;
       }
/*
* Receive results for outstanding work requests.
*/
       for (rank = 1; rank < ntasks; ++rank) {
              MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       }
/*
* Tell all the slaves to exit.
*/
       for (rank = 1; rank < ntasks; ++rank) {
              MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
       }
}

slave()
{
       double              result;
       int                 work;
       MPI_Status          status;
       for (;;) {
              MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
              MPI_COMM_WORLD, &status);
/*
* Check the tag of the received message.
*/
              if (status.MPI_TAG == DIETAG) {
                     return;
              }
              result = /* do the work */;
              MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
       }
}
```

Processes are represented by a unique rank (integer) and ranks are numbered 0, 1, 2, ..., N-1. `MPI_COMM_WORLD` means all the processes in the MPI application. It is called a communicator and it provides all information necessary to do message passing. Portable libraries do more with communicators to provide synchronisation protection that most other systems cannot handle.

15.1 Enter and Exit MPI

As with other systems, two functions are provided to initialise and clean up an MPI process:

```
       MPI_Init(&argc, &argv);
       MPI_Finalize( );
```

15.2 Who Am I? Who Are They?

Typically, a process in a parallel application needs to know who it is (its rank) and how many other processes exist. A process finds out its own rank by calling:

```
MPI_Comm_rank( ):
   Int myrank;
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
The total number of processes is returned by MPI_Comm_size( ):
   int nprocs;
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

15.3 Sending messages

A message is an array of elements of a given data type. MPI supports all the basic data types and allows a more elaborate application to construct new data types at runtime. A message is sent to a specific process and is marked by a tag (integer value) specified by the user. Tags are used to distinguish between different message types a process might send/receive. In the sample code above, the tag is used to distinguish between work and termination messages.

```
MPI_Send(buffer, count, datatype, destination, tag, MPI_COMM_WORLD);
```

15.4 Receiving messages

A receiving process specifies the tag and the rank of the sending process. `MPI_ANY_TAG` and `MPI_ANY_SOURCE` may be used optionally to receive a message of any tag and from any sending process.

```
MPI_Recv(buffer, maxcount, datatype, source, tag, MPI_COMM_WORLD, &status);
```

Information about the received message is returned in a status variable. The received message tag is status. `MPI_TAG` and the rank of the sending process is `status.MPI_SOURCE`.

Another function, not used in the sample code, returns the number of data type elements received. It is used when the number of elements received might be smaller than `maxcount`.

```
        MPI_Get_count(&status, datatype, &nelements);
```

With these few functions, you are ready to program almost any application. There are many other, more exotic functions in MPI, but all can be built upon those presented here so far.